**Name:\_\_\_\_\_Jeremiah Webb\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**M1 –Worksheet**

1. **Update the references**

Given the following link list, draw the updated list or show the changes after the execution of the provided code. If a node is eligible for garbage collection, cross out that node.

head

5

4

3

2

1

tail

1. Given the **original list**, draw the arrows for the resulting list after the following is executed:

head = head.next

head.next = head.next.next

5

4

3

2

1

1. Given the **original list**, draw the arrows for the resulting list after the following is executed:

tail.next = new Node();

tail.next.info = val

tail = tail.next;

5

4

3

2

1

1. Given the **original list**, draw the arrows for the resulting list after the following is executed:

head.next = tail;

1

2

3

4

5

1. **Implement the following functionality.**

Given a integer storing singly-linked list class **(with both a head and a tail)**, you need a method addUnique(Info val) that will only add the value to the link list if the value is not already in the list. The new value may be added to the end of the list. Implement the method from scratch as a method in the singly link list class (i.e. pretend that methods such as addToTail(Info val) or getAt(int loc) DO NOT EXIST).

e.g. If your list has the values, {5, 1, 3, 2}, addUnique(4) will result in a list with the values {5,1,3,2,4}.

However, addUnique(1) will result in {5,1,3,2}, i.e. no change. Please note, these are examples. Do not assume the content of your list.

|  |
| --- |
| public void addUnique(int val){  // Base case if head doesn't exist yet.  if(head == Null){  Node newnode = new Node();  newnode.next = Null;  newnode.info = val;  tail = newnode;  head = newnode;  return ;  }  while(head != Null){  //If the value does equal the value  if (head.info.equals(val)){  return;  }  // Cycle  head = head.next;  }  Node newnode = Node();  newnode.info = val;  newnode.next = Null;  this.tail.next = newnode;  this.tail = newnode;  } |

1. **Update references**

Original List:

a

head

tail

b

c

d

e

For each problem, start with the original list and draw the resulting list after the operations are performed. If a node is added, add it to the new list. If node(s) are deleted, then draw only the nodes and links that remain.

1. Redraw the list after performing the following operations on the original list:

Node newN = new Node();

newN.info = “f”

newN.prev = head;

newN.next = head.next.next;

head.next.next.prev = head.next;

1. Redraw the list after performing the following operations on the original list:

tail.prev.next = null;

tail = tail.prev;

1. **Write a method (in pseudocode) that deletes all instances of an integer stored within a doubly linked list with head and tail.**

|  |
| --- |
| public void deleteAll(Int val) {  Node prev = Null  Node node = this.head  while node!=Null:  if node.info.equals(val):  if prev = Null:  this.head = node.next  else:  prev.next = node.next  node.next.prev = prev  else:  prev = node  this.tail = prev |

1. **Implement addUnique() for a doubly linked list**

Given a integer storing doubly-linked list class **(with both a head and a tail)**, you need a method addUnique(int val) that will only add the value to the link list if the value is not already in the list. The new value may be added to the end of the list. Implement the method from scratch as a method in the singly link list class (i.e. pretend that methods such as addToTail(Info val) or getAt(Info val) DO NOT EXIST).

e.g. If your list has the values, {5, 1, 3, 2}, addUnique(4) will result in a list with the values {5,1,3,2,4}.

However, addUnique(1) will result in {5,1,3,2}, i.e. no change. Please note, these are examples. Do not assume the content of your list.

|  |
| --- |
| public void addUnique(Info val){  Node dummyNode = this.head;  //base case if head doesn’t exist yet  if( dummyNode == Null){  Node newnode = new Node();  newnode.next = Null;  newnode.prev = Null;  newnode.info = val;  dummyNode = newnode;  tail = newnode;  return;  }  while( dummyNode != Null){  //If the value does equal the value  if (dummyNode.info.equals(val)){  return;  }  // Cycle  dummyNode = dummyNode.next;  }  // exit out of while loop and add node  Node newnode = new Node();  newnode.info = val;  newnode.prev = this.tail  newnode.next = Null;  this.tail.next = newnode;  this.tail = newnode; |

1. **Consider a stack s containing the following integers:**

|  |
| --- |
| 4  <--top  5  1 |

Assume that the following Stack class methods are implemented:

int pop();

void push (int value);

Draw the stack s after each of the following lines of code is executed. AND, write out the value of the variable “value” whenever it is updated. Show all work. [4 points each]

b. s.push(s.pop() - 1);

|  |
| --- |
| 3  5  1 |

c. value = s.pop(); value = 7

value = 7

|  |
| --- |
| 5  1 |

d. value = s.pop();

value = 5

|  |
| --- |
| 1 |

e. s.push(s.pop());

|  |
| --- |
| 1 |

1. **Given the adder algorithm below, trace its execution as it executes. Draw out the state of each stack and the carry. Several iterations worth of blanks are provided.**

int result = add(120, 981)

|  |  |
| --- | --- |
| public static void add(int x, int y)  {  Stack s1 = new Stack<int>();  Stack s2 = new Stack<int> ();  Stack result = new Stack<int> ();  int carry = 0;  int sum = 0;  loadStacks(x, y);    //Sum Loop  while (!s1.isEmpty() && !s2.isEmpty()) {    sum = carry;    if (!s1.isEmpty())  sum += s1.pop();    if (!s2.isEmpty())  sum += s2.pop();    result.push(sum%10);  carry = (int) (sum/10);  }    //Output results  if (carry == 1)  System.out.print(carry);  while (!result.isEmpty()) {  System.out.print(result.pop());  }  } | After Load Stacks:  021  189  s1  s2  carry  result  Sum Loop - Iteration #1  s1  s2  carry  result  Sum Loop - Iteration #2  s1  s2  carry  result  Sum Loop - Iteration #3  s1  s2  carry  result  Final Result: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |

1. **Create a method that will clear the content of a stack. You may use the methods already given for stacks: push, pop, peak, isEmpty, and isFull. Implement the method such that it would work for BOTH an array-based implementation and a list-based implementation.**

/\*\*

\* Clears a stack (i.e. removes all elements)

\*

\* @param Stack s – the stack to clear (note: passed by reference)

\*/

public static void clear(Stack s)

{

//Add your code here

while(s.isFull()){

s.pop();

}

}

1. **Given the following operations, draw the resulting link-list representation of stack (you can assume either doubly linked or singly linked list, your choice).**

Stack<int> s = new ListStack<int>();

1. s.push(15);
2. s.push(13);
3. s.push(12);
4. int v2 = s.pop();
5. **Given the following operations, show the changes made to an array-based stack, which shall be initially empty with first = 0;**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | s.push(1); | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 1 |  |  |  |  |  |   First = 1 Top = 1 |
|  | int v = s.pop();  v = \_\_1\_\_\_ | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | |  |  |  |  |  |  |   First = 0 Top =0 |
|  | s.push(42) ; | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 42 |  |  |  |  |  |   First =42 Top = 42 |
|  | s.push(99); | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 42 | 99 |  |  |  |  |   First =42 Top = 99 |
|  | s.push(44); | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 42 | 99 | 44 |  |  |  |   First =42 Top = 44 |
|  | s.push(s.pop); | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 42 | 99 | 44 |  |  |  |   First =42 Top = 44 |
|  | s.push(5); | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 42 | 99 | 44 | 5 |  |  |   First =42 Top = 5 |
|  | int v2 = s.pop();  v2 = \_\_\_5\_\_\_\_\_ | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 42 | 99 | 44 |  |  |  |   First =42 Top = 44 |

// Was slightly confused by what first means, I thought it could either mean the first element in the stack, or the top of the stack as being “First” because it would be popped first. So, just in case I put both.

1. **Given the following operations, draw the resulting link-list representation of a queue (assume the list is singly linked with a first and last reference).**

Queue<int> q = new ListQueue<int>();

1. q.enqueue(15);
2. q.enqueue(13);
3. q.enqueue(12);
4. int v2 = q.dequeue();

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |

1. **Given an array-based queue implementation, redraw the array, front, and back after each of the following queue operations.**

first=0

last=0

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | q.enqueue(1) ; | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 1 |  |  |  |  |  | |
|  | int v = q.dequeue();  v = 1 | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | |  |  |  |  |  |  | |
|  | q.enqueue(2) ; | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 2 |  |  |  |  |  | |
|  | q.enqueue(3) ; | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 2 | 3 |  |  |  |  | |
|  | q.enqueue(4); | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 2 | 3 | 4 |  |  |  | |
|  | int v = q.dequeue();  v = 2 | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | |  | 3 | 4 |  |  |  | |
|  | q.enqueue(5); | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | |  | 3 | 4 | 5 |  |  | |
|  | q.enqueue(6); | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | |  | 3 | 4 | 5 | 6 |  | |

Note: For this problem, assume automatic resizing is not enabled and if the queue runs out of space it is full.

1. **Ccccombooo (combination of stack and queue)**

You must define a method that creates a new queue with the content of an original queue in reverse order. You are given a queue and a stack classes that can hold infinite amounts of data and you must use both. *Recursion is not allowed for this problem.* Your method will take as input a queue with data. Your method will output a new queue with the data in reversed order. (Note: you will not lose points for syntax errors)

Assume you have the following methods for each class:

Stack: push, pop, top, isEmpty

Queue: enqueue, dequeue, front, isEmpty

|  |
| --- |
| public Queue createReverseQueue(Queue q)  {  //Use these variables to implement the reverse  Stack s = new Stack();  Queue newQ = new Queue();  //Implement the reverse code here  // Input all items in q into a stack.  While(!q.isEmpty()){  s.push(q.dequeue());  }  while(!s.isEmpty()){  newQ.enqueue(s.pop());  }  return newQ; //return value resulting from reverse  } |